ATTNChecker: Highly-Optimized Fault Tolerant Attention for Large Language Model Training

Yuhang Liang
University of Oregon
Oregon, USA
yuhangl@uoregon.edu

Xinyi Li
Pacific Northwest National Laboratory
Washington, USA
xinyi.li@pnnl.gov

Jie Ren
College of William & Mary
Virginia, USA
jren03@wm.edu

Ang Li
Pacific Northwest National Laboratory
Washington, USA
ang.li@pnnl.gov

Bo Fang*
Pacific Northwest National Laboratory
Washington, USA
bo.fang@pnnl.gov

Jieyang Chen*
University of Oregon
Oregon, USA
jieyang@uoregon.edu

Abstract

Large Language Models (LLMs) have demonstrated remarkable performance in various natural language processing tasks. However, the training of these models is computationally intensive and susceptible to faults, particularly in the attention mechanism, which is a critical component of transformer-based LLMs. In this paper, we investigate the impact of faults on LLM training, focusing on INF, NaN, and near-INF values in the computation results with systematic fault injection experiments. We observe the propagation patterns of these errors, which can trigger non-trainable states in the model and disrupt training, forcing the procedure to load from checkpoints. To mitigate the impact of these faults, we propose ATTNChecker, the first Algorithm-Based Fault Tolerance (ABFT) technique tailored for the attention mechanism in LLMs. ATTNChecker is designed based on fault propagation patterns of LLM and incorporates performance optimization to adapt to both system reliability and model vulnerability while providing lightweight protection for fast LLM training. Evaluations on four LLMs show that ATTNChecker incurs on average 7% overhead on training while detecting and correcting all extreme errors. Compared with the state-of-the-art checkpoint/restore approach, ATTNChecker reduces recovery overhead by up to 49×.

CCS Concepts: • **Computing methodologies** → **Machine learning**; • **Computer systems organization** → **Parallel architectures**; **Dependable and fault-tolerant systems and networks**.

*Co-corresponding authors.

PPoPP '25, March 1–5, 2025, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1443-6/25/03
https://doi.org/10.1145/3710848.3710870

Keywords: Algorithm-Based Fault Tolerance, Attention Mechanism, Large Language Models, Matrix Multiplication

1 Introduction

Large Language Models (LLMs) commonly consume significant resources for training and fine-tuning. With the model consisting of more parameters and complexity, managing and updating this vast number of parameters during the training process demands substantial computational power. In addition, LLMs are trained on extensive collections of data, also consuming significant computing resources to finish the training steps. As a result, training an LLM is an extremely time and resource-intensive process, often taking weeks or even months to reach the desired accuracy. Most LLMs, therefore, are accelerated by high-performance systems featuring a massive number of GPUs, dedicated interconnects, and so on. For example, training GPT-3 [2], reportedly took thousands of GPUs. The MegaScale project [22] indicates using 10,000 GPUs to meet the demand of such large-scale training. Smaller models like BERT [10], which has around 110 million parameters for its base version, might require fewer resources but are still GPU-intensive tasks.

As systems scale up and GPUs are extensively utilized, it is risky to experience system-wise transient hardware faults, a phenomenon well-documented in the field report of large systems [11, 15, 30, 31]. Training LLMs with HPC resources needs to overcome the soft errors. The growing size and complexity of LLMs make it imperative to mitigate the impact of soft errors during training. Previous studies [46] have explored the resilience of machine learning model training against minor data deviations caused by soft errors, concluding that such deviations often result in benign outcomes, with models typically converging successfully despite these errors. A more recent investigation [19] provides a comprehensive analysis through systematic fault injection experiments conducted during model training. This study is particularly interesting as it offers insights into the implications

of specific faults, notably Infinity (INF) and Not-a-Number (NaN) errors, in the context of LLM training. Unlike normal errors, LLM training typically cannot recover from INF or NaN errors via additional training cycles.

The best practice to handle the occurrence of INF and NaN during the training is for the model to roll back to a checkpoint where no INF and NaN are present and retrain the model from the stored parameters [40]. While effective, retraining is considered a costly recovery method for LLM training in terms of time overhead, resource occupation, and expense in dollars [34, 38, 41]. Moreover, since INF and NaN can propagate during the computation, it might be necessary for the model to roll back to an earlier checkpoint that is steps away, which further exacerbates the repair cost.

To overcome the outcome of INF and NAN on the model training and resolve the inefficiency introduced by the current solution, we introduce an innovative algorithm-based fault tolerance technique (a.k.a ABFT) optimized for LLM training. The proposed technique, denoted as ATTNChecker, is designed to efficiently and effectively identify and rectify abnormal values in real-time during training. The primary objective of ATTNChecker is to develop a mechanism that is both lightweight and specifically tailored to the architectural needs of LLMs. ATTNChecker offers two advantages: First, the ability to detect anomalies where they initially occur prevents the further spread of errors through the system, thereby maintaining the integrity of the training process. Second, ABFT's capacity for immediate correction enables the system to recalibrate any infinite (INF), undefined (NaN), or other types of errors that can eventually lead to INF or NAN (i.e., near-infinite (near-INF) values) through the fast correction process. This not only relaxes the need for extensive rollback procedures but also significantly reduces the dependency on checkpoint-based recovery methods, thereby streamlining the training process and enhancing the overall training efficiency.

Several challenges need to be addressed for ATTNChecker to apply to LLMs, including: (a). Unlike normal numeric errors, INF, NaN, and near-INF would corrupt the entire checksums, which is the key technique employed in most ABFT techniques for error detection and correction. The corrupted checksums are no longer capable of locating errors and recovering the correct values; (b). It is unclear to identify the particular operations for applying ABFT, as the LLMs generally consist of distinct network architectures. A universal application of ABFT techniques might cause unnecessary computation cycles and inefficient protection; (c). Different mathematical operations usually require dedicated ABFT techniques. Given such complex operational space in LLMs, it is unclear which particular ABFT techniques are more beneficial, while a trade-off between the scope of ABFT and optimization effort should be explored.

ATTNChecker aims to address the above challenges with the following assumptions. The attention mechanisms, particularly self-attention in Transformer-based LLMs, are among the most computationally demanding components in these models. This high demand stems from the quadratic increase in computational complexity with the length of input sequences, intensive matrix multiplication operations, and the large number of parameters within the attention layers. Furthermore, the attention mechanism's frequent and intensive memory access can heavily occupy the system's memory bandwidth. These factors combined make the attention mechanism a central focus not only for the purpose of optimizing the efficiency of LLMs but also for improving the overall fault tolerance of LLM training.

Our paper makes the following contributions.

- We conduct the first comprehensive fault injection and error propagation study on the attention mechanism against INF, NaN, and near-INF errors. In addition, we conduct the first vulnerability study for the key operations in the attention mechanism given INF, NaN, and near-INF errors.
- We design the first ABFT that can effectively handle INF, NaN, and near-INF errors - Extreme Error Correcting ABFT (EEC-ABFT). EEC-ABFT is also the first ABFT that is highly optimized to handle errors in various circumstances, including propagated errors, unpredictable patterns, and mixed error types.
- Based on EEC-ABFT, we build the first comprehensive soft error protection approach for attention mechanism-ATTNChecker. ATTNChecker is specially tailored for protecting all major operations in attention and optimized for adaptive system reliability. As ATTNChecker is completely transparent to models, any existing LLMs can improve reliability by integrating our ATTNChecker with minimum modification efforts and complementary with existing fault tolerance mechanisms.
- We integrate ATTNChecker into the PyTorch framework. The evaluation shows that ATTNChecker incurs on average 7% overhead on training while reaching 100% detection and correction rate across all extreme errors. Our performance estimation shows that ATTNChecker reduces recovery overhead by up to 49× compared with the state-of-the-art recovery technique.

2 Background

2.1 Attention in LLMs

The attention mechanism is the core building block in Large Language Models (LLMs). It maps a query and a set of key-value pairs to the result of a weighted sum, effectively providing a contextually relevant representation based on the input query and the key-value pairs [37]. Query, Key, and Value are matrices, each representing instance vectors. The

Self-attention could be described as:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \quad (1)$$

where d_k is the dimension of Q and K, and $\sqrt{d_k}$ used to scale the result of $AS = QK^T$ (called Attention Scores).

Based on self-attention, the multi-head attention[37] is implemented by linearly projecting Q, K, and V multiple times, then each head performs self-attention based on its own Q, K, and V. The results of all heads will be combined. Multi-Head Attention is described as:

$$MultiHead(Q, K, V) = Concat(head_1, head_2, ..., head_n)W^o$$
$$each\ head_i = Attention(QW_i^q, KW_i^k, VW_i^v) \quad (2)$$

All matrix operations in the multi-head attention can be described using the execution flow shown in Figure 1 including six matrix multiplications and one softmax operation.

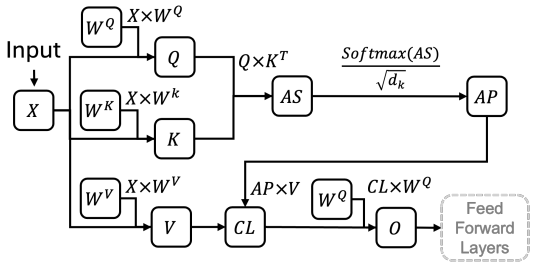

Figure 1. Execution Flow in Attention

2.2 Soft Errors causing INF, NaN, or near-INF Values

According to IEEE754, the binary representation of floating point numbers includes 3 basic parts: sign, exponent, and mantissa. Depending on where bit-flips occur they can affect the original value differently. Two specific types of floating-point errors can be caused by bit-flips: INF (i.e., infinity) and NaN (i.e., Not-a-Number). Although not classified as an exception by convention, extremely large values (near-INF) resulting from bit-flips in the exponent can also negatively impact computation. There are no specific rules for identifying near-INF, we generally consider values that are larger than a threshold as near-INF, which can vary for different computations. INF, NaN, and near-INF could cause several exceptions during the computation, as they can legally propagate through calculations e.g., INF times any number equals INF. In addition, one type of exception can transit to another type e.g., A near-INF number operating on another near-INF may lead to INF.

In a matrix multiplication (GEMM) $C = A \times B$, where A and B are input and C is the output, the presence of error elements can negatively impact the result through propagation. Figure 2 shows two examples of error propagations in GEMM. The propagation patterns can be classified as[3]:

- 0D: One standalone error without propagation. It typically appears in the origin matrix where fault strikes.

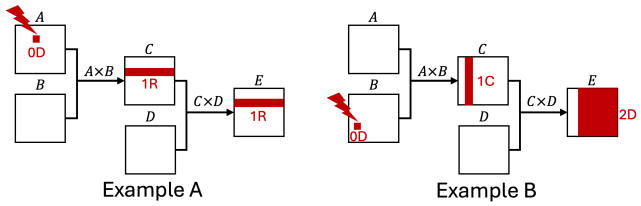

Figure 2. Two Examples of Propagation in GEMM. (A): Error Only Propagates to 1R. (B): Error Propagates to 1C then 2D.

- 1D: Errors accumulate along one row or column (entire or partial). We distinguish them using 1R for one row and 1C for one column.
- 2D: Errors accumulate beyond 1D such as a sub-matrix.

2.3 ABFT for Matrix Multiplication

ABFT [21] uses redundant information to verify the correctness of a particular algorithm under the presence of errors. For matrix operations, ABFT encodes redundant information using checksums for error protection [3, 9]. To apply ABFT on GEMM, input matrix A and B are first encoded with column checksum A^c and row checksum B^r, where a checksum is a (weighted) sum of matrix elements along columns/rows. To both detect and correct errors, two checksums are needed. A typical choice of the two checksum weights are: $v_1 = [1, 1, 1...1]^T$ (unweighted) and $v_2 = [1, 2, 3...n]^T$ (weighted). So the column and row checksums can be encoded as $A^c = \begin{bmatrix} v_1^T \\ v_2^T \end{bmatrix} \cdot A$ and $B^r = B \cdot \begin{bmatrix} v_1 v_2 \end{bmatrix}$. Next, we compute the output C and its checksums: $C^c = A^c \cdot B$ and $C^r = B \cdot A^r$ To detect errors, one needs to check if the linear relationship still holds for matrix C by recalculating the checksums ($C^{c'}$ and $C^{r'}$) and comparing with C^c and C^r. Using the column checksum correction as an example (row checksum correction is similar): we compare $C^{c'}$ with C^c to see whether they are close enough (within roundoff error E [3]) by calculating: $\delta = C^c - C^{c'}$. For instance, if $|\delta_{0,j}| > E$, then an error is detected on the j^{th} column of C. Then, $\delta_{1,j}/\delta_{0,j} = i$ (round to the nearest integer) calculates the error's row index i and $\delta_{0,j}$ is the difference between the true and corrupted value, which can be used for correction.

Existing ABFTs[3–5, 26, 43, 44] are designed to handle moderate value changes due to errors. However, they cannot efficiently handle extreme data corruptions that lead to INF, NaN, or near-INF values. This is because when locating the error, $\delta_{1,*}/\delta_{0,*}$ would be INF or NaN if the error is INF or NaN, causing incorrect error index calculation. Although not guaranteed, an error with a value close to INF (i.e., near-INF) might also lead to INF in δ due to the overflow. Given such limitations, existing ABFTs are not suitable to detect and correct errors that lead to INF, NaN, or near-INF values.

Table 1. Table of Notation

Notation	Description
Q	Query
K	Key
V	Value
AS	Attention Score
AP	Attention Probability
CL	Context Layer
O	Output

Table 2. Error Propagation Patterns in Attention Mechanism. FI: fault injecting matrix. ∞/Θ/N: INF, NaN, and near-INF. ∞*: mixture of +/−∞. M: mixture of all fault types.

		Propagating Matrix →						
		Q	K	V	AS	AP	CL	O
Inject INF(∞)	Q	**FI**	-	-	1R-∞*	1R-Θ	1R-Θ	1R-Θ
	K		**FI**	-	1C-∞*	2D-Θ	2D-Θ	2D-Θ
	V			**FI**	-	-	1C-M	2D-M
	AS				**FI**	1R-Θ	1R-Θ	1R-Θ
	CL						**FI**	1R-∞
Inject NaN(Θ)	Q	**FI**	-	-	1R-Θ	1R-Θ	1R-Θ	1R-Θ
	K		**FI**	-	1C-Θ	2D-Θ	2D-Θ	2D-Θ
	V			**FI**	-	-	1C-Θ	2D-Θ
	AS				**FI**	1R-Θ	1R-Θ	1R-Θ
	CL						**FI**	1R-Θ
Inject nINF(N)	Q	**FI**	-	-	1R-M	1R-Θ	1R-Θ	1R-Θ
	K		**FI**	-	1C-M	2D-Θ	2D-Θ	2D-Θ
	V			**FI**	-	-	1C-M	2D-M
	AS				**FI**	1R-M	1R-M	1R-M
	CL						**FI**	1R-M

3 Fault Injection and Error Propagation Study for Attention

Prior studies [19] have shown that a small perturbation affecting the LLM training data is likely to be mitigated over the model convergence process, and the final model's accuracy would be pertained under such errors. However, in terms of floating point values that belong to special categories like INF, NaN, and near-INF, prior studies have demonstrated that the LLM models would be seriously compromised under such errors, leading to unacceptable outcomes. Moreover, a near-INF number is also problematic as it can either accumulate and result in INF/NaN or disrupt the gradient distribution to reduce the convergence probability.

To understand the different impacts of soft errors on the LLM model training process, we conduct a fault injection and error propagation study on four representative LLMs: Bert, GPT-2, GPT-Neo, and Roberta. We specifically focus on studying the impact of faults that lead to INF, NaN, and near-INF since those lead to the most drastic value change that may greatly impact model training. The goal of our study is to understand (1) how an error propagates across operations in attention modules; (2) How do such errors impact the training state? We hypothesize that the answers to these questions would guide the design of efficient and effective ABFT approaches for attention modules.

Table 3. Ratios of Matrix Multiplications Workloads Relative to the Entire Attention Mechanism

Model	Bert	GPT-2	GPT-Neo	Roberta
GEMM Ratio	99.7%	99.5%	99.3%	99.7%

We select the compute-intensive GEMMs in the attention mechanism as the fault injection site since they consume most of the computation cycles for the entire training process. (Table 3 shows the workload ratios of GEMM in attention mechanism in four LLMs [20].) In particular, we inject one fault (i.e., 0D pattern) into the result matrix of each GEMM to simulate a fault that occurred during the computation. Then, we carefully examine the errors in matrices involved in the following operations. The notations are presented in Table 1

3.1 Observation summary

Table 2 summarizes the error propagation patterns in the downstream matrices following each fault injecting matrices: Q, K, V, AS, and CL inside the attention mechanism. We combine the discovery from all four LLMs as the error patterns are similar. We observe that errors can quickly propagate with diverse types of chances throughout the execution attention. In addition, we evaluate the error's impact on the training status (e.g., training loss) and compare them with fault-free executions. As the studied space is extensive, we ensure our findings are statistically significant by repeatedly injecting each type of fault. Specifically, for each GEMM, we randomly select 10% (~5,000) elements of each output matrix to increase the statistic confidence and present our findings collectively. We mainly focus on whether an error will cause a non-trainable state: a numerical data corruption that causes a loss being NaN (i.e. we only observe that the loss being NaN would lead to an untrainable state). Table 4 shows the probability of a non-trainable state caused by different error types. We observe very diverse fault sensitivity across different operations in attention.

3.2 Insight for Designing Fault Tolerance

Next, based on our error propagation study, we summarize our insights to help us design efficient and effective fault tolerance for attention mechanism in Section 4.

Segmented Protection: As error propagations could easily lead to a catastrophic result, operations in attention mechanism need to be segmented with fault detection at the operation boundary to confine the error. For example, if a 2D pattern is beyond recoverable, then error detection/correction needs to be done for AS, CL, and O to minimize the probability of the occurrence of a 2D pattern.

Handling (Non) deterministic Patterns: For operations with deterministic error patterns, efficient fault tolerance can be implemented as the deterministic pattern is bounded with no

Table 4. Probability of Non-trainable States Caused by Error

Error Type	Model	Fault Injecting Matrix				
		Q	K	V	AS	CL
INF	Bert	100%	100%	100%	100%	100%
	GPT-2	91.8%	86.8%	100%	56.9%	100%
	Neo	100%	85.6%	100.0%	54.7%	100%
	Roberta	100%	99.9%	100.0%	100%	100%
NaN	Bert	100%	100%	100%	100%	100%
	GPT-2	100%	100%	100%	54.7%	100%
	Neo	100%	100%	100.0%	54.7%	100%
	Roberta	100%	100%	100.0%	100%	100%
nINF	Bert	45.9%	43.4%	6.3%	0.2%	0.6%
	GPT-2	38.4%	37.2%	1.0%	0.5%	0.7%
	Neo	10.3%	14.4%	5.8%	11.2%	9.6%
	Roberta	54.0%	49.9%	3.6%	5.5%	0.4%

additional pattern detection required. For instance, if we use segmented fault tolerance, then O will have only deterministic 1R patterns (0D pattern can be handled the same way as 1R). However, for operations with nondeterministic error patterns, fault tolerance needs to be adaptive to different potential patterns. For example, the error pattern of AS can be either 1R or 1C, depending on the origin of the error.

Handling Mixed-type Patterns: In many cases, the propagated pattern contains a mixture of error types (e.g., +/-INF or a mixture of three types). In those cases, error detection and correcting can be challenging because different error types need to be handled differently. In addition, the detection of one error type may cause the error to transit to another type. For example, computing for detecting near-INF errors may result in INF errors. So, fault tolerance needs to be specially designed to handle mixed-type patterns and type transitions.

Selective Error Handling: The vulnerability of each operation varies across different error types and models. Higher vulnerable operation requires close attention (e.g., Q), while less vulnerable ones (e.g., CL) would likely benefit from less intensive fault tolerance methods. To balance both fault tolerance coverage and overhead, such methods must be selective against different types of errors and different models.

4 ABFT for Attention Mechanism

In this section, we propose to build the first ABFT for the attention mechanism, a key building block of many LLMs. This section introduces our fault model, which defines the types of faults ATTNChecker is interested in. Then, we introduce ABFT with an extreme error correcting algorithm (EEC-ABFT) against INF, NaN, and near-INF errors and various types of propagated errors in attention. We then use EEC-ABFT to build a systematic protection for the attention mechanism denoted as ATTNChecker. In addition to providing comprehensive protection, ATTNChecker also makes ABFT detection adaptable to system reliability and model vulnerability. Finally, we build highly-optimized GPU kernels for achieving lightweight ABFT protection for the attention mechanism. Our goal is to maximize the coverage of attention mechanism against soft errors such that the reliability of LLM training can be greatly improved while the training performance is well-maintained.

4.1 Fault Model

The occurrence of INF, NaN, and near-INF can originate from diverse sources: (1) Input-Related Faults: Certain inputs to the model can trigger floating-point exceptions during numerical operations, leading to INF/NaN values. (2) Hardware Faults: Bit-flips induced by radiation or manufacturing defects can affect the correctness of computations, generating extreme values. Note that memory is usually ECC-protected in HPC systems; hence, a memory error is beyond the scope of our paper. (3) Inappropriate Hyperparameters: The use of unsuitable hyperparameters, like an excessively large learning rate, as indicated in prior research [45], can cause a rapid increase in the loss function (known as loss explosion), resulting in INF/NaN values.

Input-related faults, likely caused by certain input data, cannot be effectively addressed using ABFT. This is because repeating the same computations during training would reproduce these exceptions. Conversely, issues related to loss explosion, often a consequence of inappropriate hyperparameters, can be dynamically mitigated through parameter adjustments. Therefore, the focus of the paper is to address the impact of transient hardware faults that lead to soft errors caused INF, NaN, or near-INF in the attention mechanism. These hardware-related errors are particularly problematic as they not only cause severe disruptions in the training process but also entail substantial overhead for correction in existing training frameworks. As attention mechanism is one the most important building blocks of many LLMs, this work targets soft errors that occur during the execution of attention mechanism.

4.2 Extreme Errors Correcting ABFT

Figure 3 shows our extreme error correcting ABFT (EEC-ABFT) handling errors in one column or row vector v in a matrix given the non-weighted checksum $csum_v$ and weighted checksum $wsum_v$. Similar to existing ABFT, EEC-ABFT first compares the updated checksums $csum_v$ and $wsum_v$ with recalculated checksums to get the difference δ_1 and δ_2, then use δ_1 to detect the error. Upon detecting an error, instead of relying on a unified error handling strategy, EEC-ABFT applies corresponding error locating and correction procedures for different cases:

Case 1: $\delta_1 < INF$ indicates all error(s) are less than INF. Then we count the number of elements that are greater than a threshold $T_{near-INF}$ for identifying near-INF. If such a number K is one, then it indicates that the error is not propagated. Next, we precisely locate the error. Normally, one can calculate the error index using δ_2/δ_1, but we must be careful

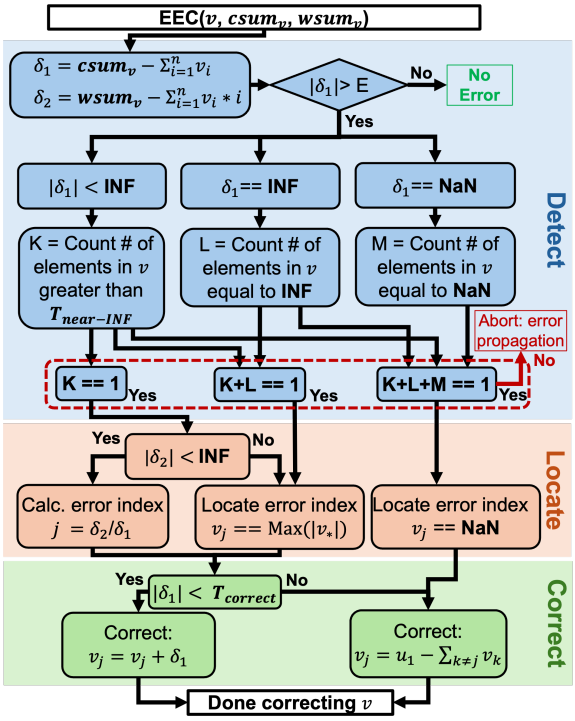

Figure 3. Extreme Error Correcting ABFT Enabling Differentiated Error Handling for INF, NaN, and near-INF Errors.

since the weighted checksum of a vector containing near-INF elements can cause overflow in δ_2. So, EEC-ABFT checks the value of δ_2 before locating the error. If $\delta_2 < INF$, we proceed with the regular locating logic. Otherwise, we search for the largest value in v to find the error index. Finally, EEC-ABFT corrects the error given the location. Special attention is worth paying to error correction for near-INF cases since a round-off error may cause a corrupted value to be wrongly corrected. In this case, δ_1 will absorb smaller v_j in the correction process leading to incorrect recovery. To avoid such precision-related errors, we define a threshold $T_{correct}$ to determine which error-correcting strategy to use. Empirically, we use $T_{near-INF} = 10^{10}$ and $T_{correct} = 10^5$.

Case 2: If $\delta_{0,i} == INF$, then it is either an INF error in the output or a near-INF error that causes δ_1 to overflow. As δ_2 may overflow, we search the max value for locating the error. To correct the error, since overflow occurs to both δ_1 and δ_2, one cannot use them to directly perform the correction. In this case, EEC-ABFT needs to reconstruct the corrupted value using existing values.

Case 3: If $\delta_i == NaN$, then all three types of errors may occur since the NAN can be caused by INF or near-INF involved computation. To locate the error, EEC-ABFT needs to search the corresponding index for NaN and use the same way as case 2 to reconstruct the corrupted value.

Case 4: For all the previous three cases, we check if there is an error propagation in v by counting the number of errors. If it is greater than one, a 1D error propagation is identified, then EEC-ABFT will abort the current correction process as

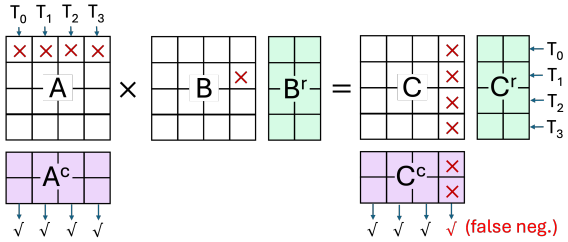

Figure 4. Deterministic (1R in A) and Nondeterministic (1C in C) Error Pattern Handling Using EEC-ABFT (T_0-T_3 represents four threads)

we need to initiate a special procedure for error propagation. (will be discussed in Section 4.3).

4.3 Correcting Extreme Error Propagations

Next, we describe how we adaptively handle three common error patterns in the attention mechanism and discuss how we efficiently parallelize the correction procedures on GPUs. Note that we limit EEC-ABFT to handle up to 1D patterns for all cases since 2D patterns cannot be efficiently corrected. We defer our discussion on propagation mitigation to Section 5, where we design systematic ABFT schemes for attention.

Deterministic Patterns: An error pattern is deterministic for an operation when only one type of propagation pattern can occur in its output. In this case, we only need to maintain the corresponding checksums for error correcting. For example, the 1C pattern only needs row checksums, and the 1R pattern only needs column checksums. The 0D pattern can be corrected using either checksum. Figure 4 shows error-correcting on two matrices: A and C. Assuming A has a deterministic 1R error, we illustrate how to detect such patterns efficiently in parallel using EEC-ABFT. Specifically, we let each thread execute EEC-ABFT on column vector in parallel on GPU. We use T_0 - T_3 to represent four threads for correcting A as an example. For fault-free execution, all executions are divergence-free on GPUs. For the 0D pattern, only the error-handling thread will execute the correction procedure, and all other threads will exit early to release the GPU resources. For 1D patterns, all threads will execute the correction procedure divergent-free when all error types are the same. Slight divergence may occur when a mixed-type pattern occurs.

Nondeterministic Patterns: Nondeterministic patterns raise two challenges: (1) the pattern is unknown in advance; (2) when corrupted elements are used for the following operation, they may not only propagate the error to the output but also corrupt the checksums of the output. For the 1C pattern, column checksums will be corrupted. For the 1R pattern, on the other hand, the row checksums will be corrupted. Figure 4 illustrates the process of recovering a nondeterministic 1D error with 1C being the true pattern and column checksum being corrupted (due to $C^c = A^c \times B$, assuming A is error-free after correction). We need to maintain both column and row checksums to handle such a pattern. To recover,

we first try using column checksums to fix potential errors. We will encounter one of two cases: (1) If non-extreme errors occur: Since the column checksums are calculated using the corrupted value, they will wrongly hold a relationship with the corrupted C, so no error is detected (false negative). (2) if extreme errors occur: EEC-ABFT will recognize and report the propagation (case 4 of Section 4.2). In either case, no error correction will be done and we will then proceed to use uncorrupted row checksums to fix the errors. Finally, we will use the corrected columns to recover the corrupted column checksums using re-computation. If a 1R pattern occurs, on the other hand, column checksums would be able to correct all errors in the first step.

Mixed-type Patterns: When a mixed-type pattern occurs, we cannot directly conclude the error type based on the value δ_1, because of error type transition. For example, near-INF in the 1D pattern can lead to δ_1 being INF, which may cause confusion for error detection. EEC-ABFT overcomes such a challenge by counting all possible error types before making conclusions about error patterns. For example, if $\delta_1 = INF$, we count both INF and near-INF. If $\delta_1 = NAN$, all three kinds of error need to be considered.

4.4 Systematic ABFT Scheme for Attention

Next, we describe how to use EEC-ABFT to build systematic ABFT protection for attention. As our EEC-ABFT can correct errors up to 1D, we divide the whole execution flow of attention mechanism into three *protection sections* to minimize the probability of more serious error propagations (e.g., 2D). Specifically, the three sections are: $S_{AS} = \{X \times W^Q, X \times W^K, Q \times K^T\}$, $S_{CL} = \{X \times W^V, AP \times V\}$, $S_O = \{CL \times W^O\}$. In addition, to ensure propagated errors can be effectively corrected and reduce error detection overhead, we carefully design a checksum passing mechanism for each section. Specifically, we only need to compute the initial checksums for the inputs of the first operation and pass the updated checksums between operations to form protection for a whole section. This enables us to correct up to 1 error at a time per section.

Attention Score Protection Section: As shown in Figure 5(a), to protect the calculation of the attention score, we encode the input matrix X with column checksums. After the multiplying input with parameter matrices, the output Q and K matrix will have their corresponding column checksums. If any computational error occurs during the multiplication, it will manifest as a 0D pattern shown as a red/green element in Q and K. Although we can use their column checksums to correct those errors, we reduce the error handling overhead by delaying detecting and correcting until $Q \times K^T$ is done. Pre-existing errors in Q or K will propagate as a 1D pattern (shown as a red/green column in AS), which is tolerable using EEC-ABFT. Finally, 0D errors that occur during $Q \times K^T$ (shown as a blue element in AS) can also be corrected immediately after computing AS.

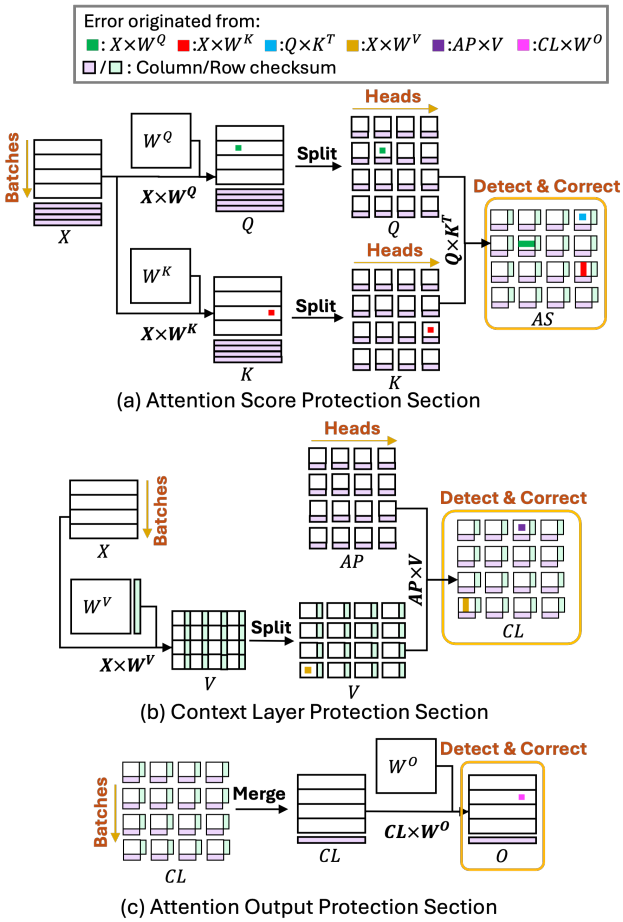

Figure 5. Systematic ABFT Schemes in ATTNChecker. All 6 GEMMs Form Three Protection Sections

Context Layer Protection Section: Figure 5(b) shows our protection mechanism for the calculation of CL. We encode the parameter matrix W^V with row checksums. By doing so, V will have row checksums encoded after $X \times W^V$. Next, we encode AP with column checksum so that after multiplying AP with V, the resulting CL will have both row and column checksums. If errors occur during $X \times W^V$, it will manifest as a 0D error pattern shown as a yellow element in V. Similarly, we delay the error detection until CL is computed. Therefore, any 0D error in V will manifest as a 1D error in CL, which can be corrected using its checksums. In addition, 0D errors that occur during $AP \times V$ (shown as purple elements in CL) can be corrected using the checksums of CL.

Attention Output Protection Section: Figure 5(c) shows the protection on the attention output O. Since it only involves one matrix multiplication, we only need to maintain one side checksum. As we already maintain the column checksums for CL after $AP \times V$, we choose to keep the column checksums and update them along with $CL \times W^O$. The resulting output matrix O will have column checksums that can detect and correct the 0D errors (shown as magenta elements in O) that occurred during $CL \times W^O$.

4.5 Adaptive ABFT Detection Frequencies

To make ATTNChecker adaptive to operations with different vulnerabilities in attention mechanism, we aim to optimize ABFT's detection frequency. Assuming a system has an error rate (measured by the number of errors per flop that lead to INF, NaN, or near-INF): λ_{INF}, λ_{NaN}, and λ_{nINF}, we treat the probability distribution of number of error occurrences as the Poisson distribution[6]. So, the probability of having k errors of type e in an operation OP with total n_{OP} flops can be estimated using $P^E_{OP}(k) = \frac{e^{-\lambda_E \times n_{OP}}(\lambda_E \times n_{OP})^k}{k!}$. Then given protection section $S = \{OP_1, OP_2, ..., OP_m\}$ and a set of error types $E = INF, NaN, near-INF$, then the probability of having no error in the section is $R^{free}_S = \prod_{i=1}^{m} \prod_{e \in E} P^e_{OP_i}(0)$. Similarly, the probability of having exactly only one error of type e in OP_j and no error for other operations in S is:

$$R^{INF}_S(j) = P^{INF}_{OP_j}(1) P^{NaN}_{OP_j}(0) P^{nINF}_{OP_j}(0) R^{free}_{S \setminus OP_j}$$
$$R^{NaN}_S(j) = P^{INF}_{OP_j}(0) P^{NaN}_{OP_j}(1) P^{nINF}_{OP_j}(0) R^{free}_{S \setminus OP_j}$$
$$R^{nINF}_S(j) = P^{INF}_{OP_j}(0) P^{NaN}_{OP_j}(0) P^{nINF}_{OP_j}(1) R^{free}_{S \setminus OP_j}$$

Based on our ABFT scheme design, each segment can tolerate up to one error that occurs in any operation. As an error does not always lead to non-trainable states, we quantify the vulnerability of an operation by defining ϕ^T_{OP} being the probability of a type e error in OP that leads to a non-trainable state, which can be estimated by profiling (Table 4). Assuming f_S represents the frequency at which ABFT is enabled for the entire series S (e.g., f_S = 0.5 corresponds to enabling ABFT every other time we execute S), we define $H^e_i = f_S + (1 - f_S)\phi^e_{OP_i}$. This expression represents the probability of encountering a type e error that can be either correctly handled by ABFT or not handled but does not result in a non-trainable state. So, we define a quantified estimation of the probability that ABFT can correct all errors in S - fault coverage (FC).

$$FC_S = R^{free}_S + \sum_{i=1}^{m} \sum_{e \in E} R^e_S(i) H^e_i$$

The FC for the attention mechanism can be estimated as:

$$FC_{att}(f_{AS}, f_{CL}, f_O) = FC_{AS} FC_{CL} FC_O$$

To optimize the ABFT frequencies, we define T_S as the ABFT overhead the ABFT is added to protect the section S. Given a target FC_{target} for attention mechanism (e.g., up to 1 error per billion times of execution), finding the optimized frequency can be converted into an optimization problem:

$$\text{Minimize } T = f_{AS} T_{AS} + f_{CL} T_{CL} + f_O T_O$$
$$\text{while } FC_{att}(f_{AS}, f_{CL}, f_O) \geq FC_{target}$$

To solve the above optimization problem, we introduce a term - fault coverage efficiency (FCE) - to estimate how much FC is gained with a time unit of extra ABFT overhead paid for protecting a section S:

$$FCE_S = \frac{\partial FC_S}{\partial T} = \frac{R^{free}_S + \sum_{i=1}^{m} \sum_{e \in E} R^e_S(i)(1 - \phi^e_{OP_i})}{T_S}$$

Then, finding the optimized ABFT frequencies can be solved using a greedy-based algorithm as shown in Algorithm 1. The algorithm first computes FCE for all sections and sorts them in descending order. Then it loops over all sections starting from the most efficient to the least efficient and allocates t_S time for protecting S. The maximum time we can allocate is T_S. The selection logic is to select as much more efficient protection as possible. Finally, we convert time allocations for each section into protection frequencies.

Algorithm 1: ABFT Frequencies Optimization

```
Function OptimizeABFTFrequencies(λ^INF, λ^NaN, λ^nINF, φ, FC_target):
    FCE_AS, FCE_CL, FCE_O ← λ^INF, λ^NaN, λ^nINF, φ
    FC ← 0 ; t_AS, t_CL, t_O ← 0
    for S in DescendSort(FCE) do
        if FC_target − FC < FCE_S T_S then
            t_S = T_S
        end
        if FC < FC_target then
            t_S = (FC_target − FC) / FCE_S
        end
        FC ← FC + FCE_S t_S
    end
    f_AS ← t_AS / T_AS ; f_CL ← t_CL / T_CL ; f_O ← t_O / T_O
return f_AS, f_CL, f_O
```

4.6 Performance Optimization for GPUs

Encoding: Enabling ABFT for batched matrix operations (e.g., $K \times V$) creates an unique computation pattern. This pattern is not well-optimized in vendor-optimized libraries such as NVIDIA cuBLAS. We design highly customized and efficient kernels for encoding checksums to overcome this. In our kernel, we parallelize the encoding process along the Streaming Multiprocessor by the Number of Heads × Number of Batches to improve GPU occupancy. To encode each block, we load all data into shared memory and decoupled thread-data mapping between loading and computing [7, 8] such that we can achieve fully coalesced global memory accesses while minimizing bank conflict in shared memory.

Updating: To accelerate the checksum updating operation on GPUs, we pack the checksum with the operand matrix such that the checksum can be updated together with the original operation. We measure the efficiency of updating checksums as fused computation to improve GPU utilization and avoid kernel launch overhead in Section 5.3

Detection and Correction: We also build customized error detection and correction kernels for GPU. As shown in Figure 4, error detection and correction are fully parallelizable and divergence-free when no errors occur, introducing minimal overhead to the attention mechanism.

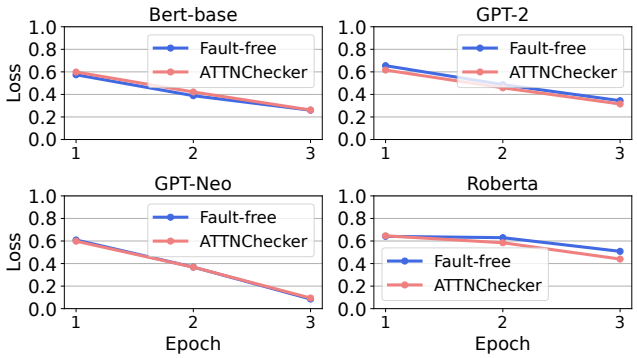

Figure 6. Training Loss of Fault-free Execution vs. Faulty Execution Recovered with ATTNChecker

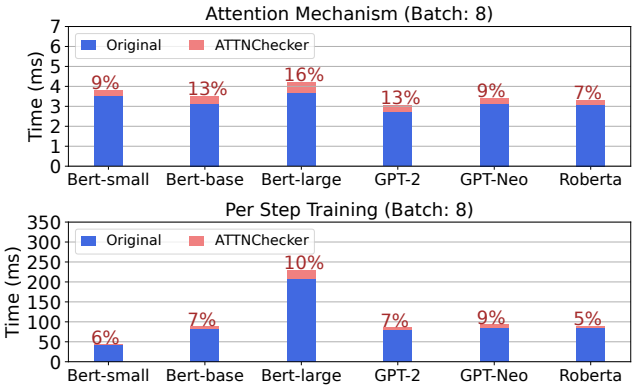

Figure 7. ATTNChecker Overhead on 6 LLMs

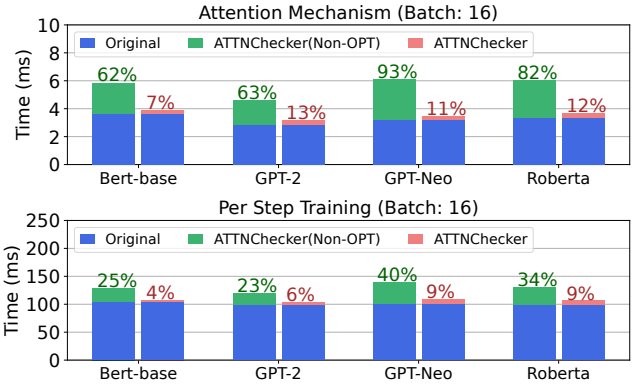

Figure 8. ATTNChecker Overhead With and Without GPU Optimization

5 Experimental Evaluation

5.1 Experiment Methodology

Evaluation Implementation: We evaluate ATTNChecker by integrating it with the latest PyTorch library[32]. For the best computing efficiency, all core ABFT algorithms are implemented using CUDA and C++ with a few extra control logic added to the Pytorch interface for enabling the functionalities of ABFT in the training process.

Models and Datasets: To demonstrate the generality of ATTNChecker, four LLMs are used in our experiments: Bert[10], GPT-2[33], GPT-Neo[14], and Roberta[28]. To evaluate the training process, we conduct fine-tuning on these models on the Microsoft Research Paraphrase Corpus (MRPC) dataset from the General Language Understanding Evaluation (GLUE) benchmark[39] for sequence classification.

Fault Injection: To estimate how effective ATTNChecker is, we inject faults via source code instrumentation. We randomly select positions in the output matrix to inject faults to simulate faults that occur during an operation. INF and NaN are injected via assignments and near-INF is injected by flipping the most significant bit of the selected element.

Experiment Platform: Our modified PyTorch is built with CUDA 12.5 and GCC 11. All evaluations are done on a Lambda GPU server equipped with 4 NVIDIA A100 GPUs with 80 GB memory on each GPU and one AMD EPYC 7513 32-core CPU with 1 TB memory. We further simulate the performance of training multi-billion parameter LLMs on large-scale multi-GPU systems.

5.2 Error Detection and Correction Capability

We first evaluate the effectiveness of ATTNChecker by studying its correct rate when different types of faults are injected at different stages of the attention mechanism. Each time, we inject one error per execution to a random position of a matrix in the attention mechanism with ABFT enabled. We repeat our fault injection such that we inject a total of 10% (~5,000) elements of each output matrix in the attention mechanism. Our evaluation of four LLMs shows that **all errors can be detected and successfully corrected** to their original values. Additionally, Figure 6 shows that ATTNChecker makes a negligible impact on the training loss after error recovery.

5.3 ABFT Overhead

Next, we evaluate the computational overhead caused by ATTNChecker for both attention mechanism and the end-to-end training process. Figure 7 shows the overhead results across all four models, including three different sizes of the Bert model. In particular, ATTNChecker incurs about 11% overhead to attention mechanism block along and 7% overhead for the entire training step on average. Figure 8 shows the overhead of attention mechanism for training with and without performance optimization for GPUs. Our dedicated optimization on GPU reduces ABFT overhead by up to 8.6× for the attention mechanism and 6.0× for training. Furthermore, comparing results of different batch sizes (8 and 16) shows that ATTNChecker bring similar overhead across different batch sizes (we only show the result of batch size = 16 due to page limit). Figure 9 shows the checksum encoding throughputs using cuBLAS 12.5 and ATTNChecker's optimized encoder kernel. Our optimized kernel outperforms cuBLAS by 13× with up to 91.4% memory bandwidth utilization, while cuBLAS only achieves less than 10%.

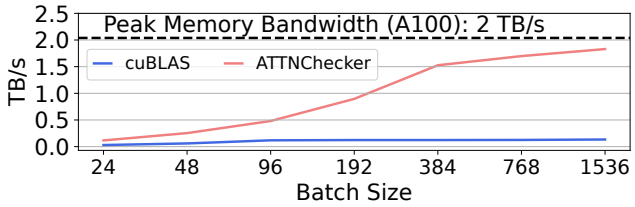

Figure 9. Checksum Encoding Throughput

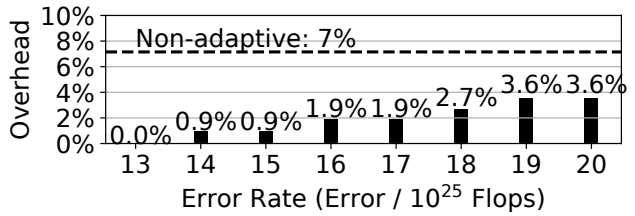

Figure 10. Training Overhead Reduction With Optimized ABFT Detection Frequencies

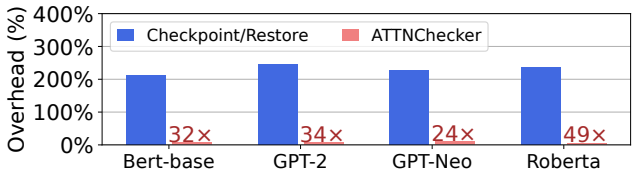

Figure 11. Per Training Step Recovery Overhead (Checkpoint/Restore vs. ATTNChecker)

5.4 Adaptive ABFT Detection Frequencies

To demonstrate how ATTNChecker can be adaptive to various error rates representing a wider spectrum of systems, we use our adaptive ABFT optimization algorithm to optimize ABFT detection frequencies for the three detection sections: S_{AS}, S_{CL}, and S_O. We assume the system error rate varies from 13 to 20 errors per 10^{25} flops for all three types of error according to the latest field report [12]. Our target fault coverage is 1 failure per 10^{11} executions of attention mechanism based on the operation vulnerability of the Bert model. Figure 10 shows the per-step training overhead with optimized ABFT checking frequencies. When the error rate is small, we can see that most of the ABFT overhead can be reduced (even to 0 when ABFT is not needed). As the error rate increases, ABFT overhead steadily increases with the detection frequencies yet remains minimal (less than 3.6%).

5.5 Recovery Overhead

In addition to detection, error correction brings extra overhead. Our evaluation shows that the correcting 1D error caused by error propagation from Q, K, and V incurs 0.7% overhead on average. 0D errors can be corrected with 0.3% overhead on average. Correcting errors in O brings 3.9% overhead on average since unlike matrices such as AS where error correction is done within a small block, O is a larger matrix merged from smaller blocks, which requires potentially higher recovery cost.

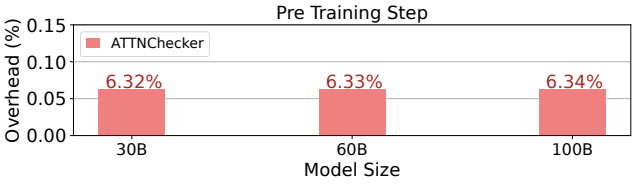

Figure 12. Overhead of ATTNChecker with multi-billion parameter LLMs on multi-GPU systems.

Figure 11 shows the pre-training step error recovery overhead of ATTNChecker compared with the checkpoint/restore (CR) approach. Specifically, we assume checkpointing is done per training step and restore is done once the model encounters a non-trainable state. Since the CR requires loading and re-executing the current training step, the CR causes more than 200% overhead once a non-trainable state is encountered. ATTNChecker, on the other hand, for correction-only introduces less than 10% overhead, which is up to 49× overhead reduction compared with CR.

5.6 ATTNChecker in Scale

To further quantify the performance impact of ATTNChecker, we simulate the training of multi-billion parameter LLMs on multi-GPU systems using the same simulation methodology as existing work [27]. Figure 12 presents the overhead of ATTNChecker when training LLMs with 30B, 60B, and 100B parameters on 1,024 GPUs using data parallelism. The results demonstrate that the ABFT overhead remains nearly constant as the number of parameters increases, highlighting the scalability and applicability of ATTNChecker for large-scale LLM training.

6 Related Works

Soft error detection. Existing works have explored various techniques to detect and mitigate the impact of soft errors in computing systems. For example, some efforts [17, 18, 36] investigate instruction-level duplication, which duplicates a subset of critical or error-prone instructions to detect soft errors affecting computations. Others [13, 16, 24] prioritize the error-proneness of applications or instructions and select the top ones for duplication or protection, identifying the most critical parts of the computation likely to cause silent data corruptions if affected by soft errors. However, these error detection methods are impractical for detecting soft errors in ML models due to their large runtime overhead.

Algorithm-Based Fault Tolerance (ABFT) techniques, on the other hand, are tailored to specific algorithms, leveraging their inherent properties to detect and correct various types of errors, including soft errors, without requiring extensive modifications to the underlying hardware or software. ABFT has been applied to protect and handle errors in matrix decomposition [21, 42, 44], and matrix multiplications [3–5, 9]. As ABFT can effectively protect matrix multiplications, several works have extended it to protect ML models such as

CNN [47] and Deep learning recommendation systems [25]. ATTNChecker also leverages ABFT but focuses on more computationally intensive and model-quality-critical attention computation in LLMs.

Resilience of ML models under transient faults. Understanding and evaluating the vulnerability of ML models to hardware faults is becoming increasingly important as ML models scale [11]. Existing work investigates various tools and metrics to identify, mitigate, and quantify the impact of faults in ML systems. Ares [35] and PyTorchFI [29] are fault injection tools built on PyTorch to help quantify model accuracy drop with faults in ML models. Li et al. [23] and Agarwal et al. [1] investigate the Silent Data Corruption (SDC) rate for CNNs and LLMs, respectively, using fault injection. While these works provide valuable insights into the resilience of ML models under transient faults, they primarily focus on fault injection and quantifying the impact of faults. In contrast, ATTNChecker offers a lightweight, real-time solution for not only detecting but also correcting errors during LLM training.

7 Conclusion

The growing complexity and size of LLMs demand effective fault tolerance to maintain efficient and reliable training. Traditional fault tolerance mechanisms such as checkpointing are resource-intensive and time-consuming, hindering the efficiency of LLM training. In response, we introduce ATTNChecker, an algorithm-based fault tolerance technique that offers a lightweight, real-time solution for detecting and correcting errors during LLM training. ATTNChecker significantly reduces recovery overheads by up to 49× compared to traditional methods, while only adding negligible overhead to the training process. By integrating ATTNChecker into the PyTorch library and testing on various LLMs, we demonstrate its efficacy in enhancing the resilience and efficiency of LLM training. This advancement provides a robust framework for future fault tolerance in large-scale machine learning models.

Acknowledgment

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, ComPort: Rigorous Testing Methods to Safeguard Software Porting, under Award Number 78284. The platforms used for evaluation in this work are supported by by the U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, under award 66150: "CENATE - Center for Advanced Architecture Evaluation". The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under Contract DE-AC05-76RL01830.

References

[1] Udit Kumar Agarwal, Abraham Chan, and Karthik Pattabiraman. 2023. Resilience Assessment of Large Language Models under Transient Hardware Faults. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 659–670.

[2] Tom B Brown. 2020. Language models are few-shot learners. *arXiv preprint ArXiv:2005.14165* (2020).

[3] Jieyang Chen, Hongbo Li, Sihuan Li, Xin Liang, Panruo Wu, Dingwen Tao, Kaiming Ouyang, Yuanlai Liu, Kai Zhao, Qiang Guan, et al. 2018. Fault tolerant one-sided matrix decompositions on heterogeneous systems with gpus. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 854–865.

[4] Jieyang Chen, Sihuan Li, and Zizhong Chen. 2016. Gpu-abft: Optimizing algorithm-based fault tolerance for heterogeneous systems with gpus. In *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE, 1–2.

[5] Jieyang Chen, Xin Liang, and Zizhong Chen. 2016. Online algorithm-based fault tolerance for cholesky decomposition on heterogeneous systems with gpus. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 993–1002.

[6] Jieyang Chen, Xin Liang, Kai Zhao, Hadi Zamani Sabzi, Laxmi Bhuyan, and Zizhong Chen. 2023. Improving energy saving of one-sided matrix decompositions on cpu-gpu heterogeneous systems. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 274–287.

[7] Jieyang Chen, Lipeng Wan, Xin Liang, Ben Whitney, Qing Liu, David Pugmire, Nicholas Thompson, Jong Youl Choi, Matthew Wolf, Todd Munson, et al. 2021. Accelerating multigrid-based hierarchical scientific data refactoring on gpus. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 859–868.

[8] Jieyang Chen, Nan Xiong, Xin Liang, Dingwen Tao, Sihuan Li, Kaiming Ouyang, Kai Zhao, Nathan DeBardeleben, Qiang Guan, and Zizhong Chen. 2019. TSM2: optimizing tall-and-skinny matrix-matrix multiplication on GPUs. In *Proceedings of the ACM International Conference on Supercomputing*. 106–116.

[9] Zizhong Chen and Jack Dongarra. 2008. Algorithm-based fault tolerance for fail-stop failures. *IEEE Transactions on Parallel and Distributed Systems* 19, 12 (2008), 1628–1641.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[11] Harish Dattatraya Dixit, Laura Boyle, Gautham Vunnam, Sneha Pendharkar, Matt Beadon, and Sriram Sankar. 2022. Detecting silent data corruptions in the wild. arXiv:2203.08989 [cs.AR] https://arxiv.org/abs/2203.08989

[12] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[13] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. 2016. ePVF: An Enhanced Program Vulnerability Factor Methodology for Cross-Layer Resilience Analysis. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), acceptance rate = 21%*. 168–179. https://doi.org/10.1109/DSN.2016.24

[14] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. 2020. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. *arXiv preprint arXiv:2101.00027* (2020).

[15] Daniel Alfonso Gonçalves Gonçalves de Oliveira, Laercio Lima Pilla, Thiago Santini, and Paolo Rech. 2016. Evaluation and Mitigation of Radiation-Induced Soft Errors in Graphics Processing Units. *IEEE Trans. Comput.* 65, 3 (2016), 791–804. https://doi.org/10.1109/TC.2015.2444855

[16] Luanzheng Guo, Dong Li, Ignacio Laguna, and Martin Schulz. 2018. FlipTracker: Understanding Natural Error Resilience in HPC Applications. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 94–107. https://doi.org/10.1109/SC.2018.00011

[17] Siva Kumar Sastry Hari, Sarita V Adve, and Helia Naeimi. 2012. Low-cost program-level detectors for reducing silent data corruptions. In *2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 1–12.

[18] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. 2012. Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[19] Yi He, Mike Hutton, Steven Chan, Robert De Gruijl, Rama Govindaraju, Nishant Patil, and Yanjing Li. 2023. Understanding and mitigating hardware failures in deep learning training systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–16.

[20] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. 2022. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556* (2022).

[21] Kuang-Hua Huang and Jacob A Abraham. 1984. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers* 100, 6 (1984), 518–528.

[22] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. 2024. {MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 745–760.

[23] Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W Keckler. 2017. Understanding error propagation in deep learning neural network (DNN) accelerators and applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[24] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai. 2018. Modeling Soft-Error Propagation in Programs. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 27–38. https://doi.org/10.1109/DSN.2018.00016

[25] Sihuan Li, Jianyu Huang, Ping Tak Peter Tang, Daya Khudia, Jongsoo Park, Harish Dattatraya Dixit, and Zizhong Chen. 2022. Efficient soft-error detection for low-precision deep learning recommendation models. In *2022 IEEE International Conference on Big Data (Big Data)*. IEEE, 1556–1563.

[26] Xin Liang, Jieyang Chen, Dingwen Tao, Sihuan Li, Panruo Wu, Hongbo Li, Kaiming Ouyang, Yuanlai Liu, Fengguang Song, and Zizhong Chen. 2017. Correcting soft errors online in fast fourier transform. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[27] Zhongyi Lin, Ning Sun, Pallab Bhattacharya, Xizhou Feng, Louis Feng, and John D Owens. 2024. Towards Universal Performance Modeling for Machine Learning Training on Multi-GPU Platforms. *arXiv preprint arXiv:2404.12674* (2024).

[28] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[29] Abdulrahman Mahmoud, Neeraj Aggarwal, Alex Nobbe, Jose Rodrigo Sanchez Vicarte, Sarita V. Adve, Christopher W. Fletcher, Iuri Frosio, and Siva Kumar Sastry Hari. 2020. PyTorchFI: A Runtime Perturbation Tool for DNNs. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 25–31. https://doi.org/10.1109/DSN-W50199.2020.00014

[30] Catello Di Martino, William Kramer, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2015. Measuring and Understanding Extreme-Scale Application Resilience: A Field Study of 5,000,000 HPC Application Runs *(Proceedings of the International Conference on Dependable Systems and Networks)*. IEEE Computer Society, 25–36. https://doi.org/10.1109/DSN.2015.50

[31] Bin Nie, Devesh Tiwari, Saurabh Gupta, Evgenia Smirni, and James H. Rogers. 2016. A large-scale study of soft-errors on GPUs in the field. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 519–530. https://doi.org/10.1109/HPCA.2016.7446091

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[33] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[34] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[35] Brandon Reagen, Udit Gupta, Lillian Pentecost, Paul Whatmough, Sae Kyu Lee, Niamh Mulholland, David Brooks, and Gu-Yeon Wei. 2018. Ares: A framework for quantifying the resilience of deep neural networks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC.2018.8465834

[36] G A Reis, J Chang, N Vachharajani, R Rangan, and D I August. 2005. SWIFT: Software Implemented Fault Tolerance. In *International Symposium on Code Generation and Optimization*. IEEE, 243–254.

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[38] Borui Wan, Mingji Han, Yiyao Sheng, Zhichao Lai, Mofan Zhang, Junda Zhang, Yanghua Peng, Haibin Lin, Xin Liu, and Chuan Wu. 2024. ByteCheckpoint: A Unified Checkpointing System for LLM Development. *arXiv preprint arXiv:2407.20143* (2024).

[39] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2018. GLUE: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461* (2018).

[40] Yuxin Wang, Shaohuai Shi, Xin He, Zhenheng Tang, Xinglin Pan, Yang Zheng, Xiaoyu Wu, Amelie Chi Zhou, Bingsheng He, and Xiaowen Chu. 2023. Reliable and Efficient In-Memory Fault Tolerance of Large Language Model Pretraining. *arXiv preprint arXiv:2310.12670* (2023).

[41] Baodong Wu, Lei Xia, Qingping Li, Kangyu Li, Xu Chen, Yongqiang Guo, Tieyao Xiang, Yuheng Chen, and Shigang Li. 2023. Transom: An efficient fault-tolerant system for training llms. *arXiv preprint arXiv:2310.10046* (2023).

[42] Panruo Wu and Zizhong Chen. 2014. FT-ScaLAPACK: Correcting soft errors on-line for ScaLAPACK Cholesky, QR, and LU factorization routines. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 49–60.

[43] Panruo Wu, Nathan DeBardeleben, Qiang Guan, Sean Blanchard, Jieyang Chen, Dingwen Tao, Xin Liang, Kaiming Ouyang, and Zizhong Chen. 2017. Silent data corruption resilient two-sided matrix factorizations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 415–427.

[44] Panruo Wu, Qiang Guan, Nathan DeBardeleben, Sean Blanchard, Dingwen Tao, Xin Liang, Jieyang Chen, and Zizhong Chen. 2016. Towards

practical algorithm based fault tolerance in dense linear algebra. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. 31–42.

[45] Chunmei Xu, Shengheng Liu, Zhaohui Yang, Yongming Huang, and Kai-Kit Wong. 2021. Learning rate optimization for federated learning exploiting over-the-air computation. *IEEE Journal on Selected Areas in Communications* 39, 12 (2021), 3742–3756.

[46] Zhao Zhang, Lei Huang, Ruizhu Huang, Weijia Xu, and Daniel S Katz. 2019. Quantifying the impact of memory errors in deep learning. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 1–12.

[47] Kai Zhao, Sheng Di, Sihuan Li, Xin Liang, Yujia Zhai, Jieyang Chen, Kaiming Ouyang, Franck Cappello, and Zizhong Chen. 2020. FT-CNN: Algorithm-based fault tolerance for convolutional neural networks. *IEEE Transactions on Parallel and Distributed Systems* 32, 7 (2020), 1677–1689.

A Artifact Appendix

This artifact contains instructions for how to replicate the experiments. It is available at: zenodo.org/records/14503617.

A.1 Evaluation environment

- Anaconda virtual environment with python 3.8.10
- CUDA Toolkit 12.6
- gcc 11.4.0
- Nividia A100 - 80GB

A.2 Build and Config

Choose one of the two build options

A.2.1 Option 1: Docker Image. Pull and run a pre-built docker image from Docker Hub.

```
$ docker pull lyh911/attnchk-pytorch:2.0
$ docker run --ipc=host --shm-size=512m --gpus all
     -it --rm lyh911/attnchk-pytorch:2.0
```

A.2.2 Option 2: Build from source. 1. Create an Anaconda environment.

```
$ conda create --name attnchk python==3.8.10
$ conda activate attnchk
```

2. Download the source code.

```
$ git clone https://github.com/liangyh911/
    ATTNChecker.git
$ cd ATTNChecker
```

3. Install the required Python packages for ATTNChecker.

```
$ pip install -r requirements.txt
```

4. Move the modeling scripts to Huggingface Transformers package.

```
$ cp ./HF-moding/modeling_bert.py <
    Path_To_Anaconda>/envs/attnchk/lib/python3.8/
    site-packages/transformers/models/bert/
    modeling_bert.py
$ cp ./HF-moding/modeling_gpt2.py      <
    Path_To_Anaconda>/envs/attnchk/lib/python3.8/
    site-packages/transformers/models/gpt2/
    modeling_gpt2.py
$ cp ./HF-moding/modeling_gpt_neo.py   <
    Path_To_Anaconda>/envs/attnchk/lib/python3.8/
    site-packages/transformers/models/gpt_neo/
    modeling_gpt_neo.py
$ cp ./HF-moding/modeling_roberta.py   <
    Path_To_Anaconda>/envs/attnchk/lib/python3.8/
    site-packages/transformers/models/roberta/
    modeling_roberta.py
```

5. Build PyTorch from the source.

```
$ cd ./ATTNChecker/pytorch
$ conda install cmake ninja
$ pip install mkl-static mkl-include
$ pip install -r requirements.txt
$ git submodule sync
$ git submodule update --init --recursive
$ export CMAKE_PREFIX_PATH=${CONDA_PREFIX:-"$(
    dirname␣$(which␣conda))/../"}
$ python setup.py develop
```

If Python builds successfully, you will see in the last line of the output:

```
Finished processing dependencies for torch==2.3.0
    a0+git83e4d29
```

Test if CUDA is available by using the following codes. *True* will be returned if cuda is available.

```
$ import torch
$ torch.cuda.is_available()
```

A.3 Measuring ATTNChecker running overhead

Use *./records/cleanRecords.py* to clean the existing records before running new scripts.

```
$ cd ./ATTNChecker
$ python ./records/cleanRecords.py
```

To measure the overhead of ATTNChecker, use the scripts in *./ABFT_running_time* folder. The default settings are

- batch size: 8
- number of training: 20

Before running the scripts, make sure to use one GPU if you have multiple GPUs on your device. Using export command.

```
$ export CUDA_VISIBLE_DEVICES=0
```

Clean the previous records before running each test.

```
# bert
$ python ./records/cleanRecords.py
$ python ./ABFT_running_time/bertTest.py
# gpt2
$ python ./records/cleanRecords.py
$ python ./ABFT_running_time/gpt2.py
# gpt neo
$ python ./records/cleanRecords.py
$ python ./ABFT_running_time/gpt-neo.py
# roberta
$ python ./records/cleanRecords.py
$ python ./ABFT_running_time/roberta.py
```

Here is an example output of the BERT model. For each test, the output results may vary. (The loss here is the 1-step training loss.)

```
Attention Mechanism Overhead:  0.14775651309451615
Training Overhead:  0.0552227860653921
ATTNChecker Loss:  0.5106
no ATTNChecker Loss:  0.5106
```

A.4 Measuring checkpoint save and load overhead

Before measuring the save and load overhead of the checkpoint, please ensure you have tested ATTNChecker Running Overhead.

Use the scripts in *./Checkpoint_time* folder to test the save and load time of checkpointing a model. Here is an example.

```
# bert
$ python ./records/cleanRecords.py
$ python ./Checkpoint_time/bert.py
# gpt2
$ python ./records/cleanRecords.py
$ python ./Checkpoint_time/gpt2.py
# gpt neo
$ python ./records/cleanRecords.py
$ python ./Checkpoint_time/gpt-neo.py
# roberta
$ python ./records/cleanRecords.py
$ python ./Checkpoint_time/roberta.py
```

Here is an example output of the BERT model. For each test, the output results may vary. The overhead is calculated in the same way as ATTNChecker running overhead.

```
Overhead of Checkpointing:  8.522816600251735
```

A.5 Measuring training loss of ATTNChecker and baseline during 3 epochs

```
# bert
$ python ./records/cleanRecords.py
$ python ./ABFT_epoch_loss/bert.py
# gpt2
$ python ./records/cleanRecords.py
$ python ./ABFT_epoch_loss/gpt2.py
# gpt neo
$ python ./records/cleanRecords.py
$ python ./ABFT_epoch_loss/gpt-neo.py
# roberta
$ python ./records/cleanRecords.py
$ python ./ABFT_epoch_loss/roberta.py
```

Here is an example output of the BERT model. For each test, the output results may vary. The baseline is training without ATTNChecker.

```
Loss of ATTNChecker:
1st epoch:  0.5349 , 2nd epoch:  0.3071 , 3rd
    epoch:  0.1285
Loss of Baseline:
1st epoch:  0.5635 , 2nd epoch:  0.3362 , 3rd
    epoch:  0.1312
```